PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 17/30, 17/40 — A2

(11) International Publication Number: WO 95/08809

(43) International Publication Date: 30 March 1995 (30.03.95)

(21) International Application Number: PCT/US94/10093

(22) International Filing Date: 9 September 1994 (09.09.94)

(30) Priority Data:
126,586 24 September 1993 (24.09.93) US

(71) Applicant: ORACLE CORPORATION [US/US]; 500 Oracle Parkway, Redwood City, CA 94065 (US).

(72) Inventors: JAIN, Sandeep; 2312 Wooster Avenue, Belmont, CA 94002 (US). DANIELS, Dean; 44990 Lynx Drive, Fremont, CA 94539 (US).

(74) Agents: HECKER, Gary, A. et al.; Hecker & Harriman, Suite 1200, 2049 Century Park East, Los Angeles, CA 90067 (US).

(81) Designated States: AM, AT, AU, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, ES, FI, GB, GE, HU, JP, KE, KG, KP, KR, KZ, LK, LR, LT, LU, LV, MD, MG, MN, MW, NL, NO, NZ, PL, PT, RO, RU, SD, SE, SI, SK, TJ, TT, UA, UZ, VN, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG), ARIPO patent (KE, MW, SD).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: METHOD AND APPARATUS FOR DATA REPLICATION

(57) Abstract

The present invention provides the ability to replicate modifications made at a local site to multiple remote sites in a peer-to-peer environment. Information regarding these replicated modifications (e.g., insert, delete, or update) are contained in a set of replication tables. Thus, modifications can be duplicated at other sites immediately after the original modification, or deferred until the remote site is available. The replication tables of the present invention include a transactions table, transaction nodes table, calls table, call nodes table, and an exceptions table. The present invention further provides a logic-oriented procedure-level replication. Procedure-level replication modifies a remote site based on the logical operations used to modify the data at the originating site. Procedure-level replication provides the ability to identify conflicting updates as well. Information concerning conflicts identified by the present invention can be retained in the replication tables. The information contained in the replication tables can be used immediately, or subsequently, to address any conflicts detected by the present invention. The present invention provides the ability to roll back any modifications made once a conflict is identified. Further, the present invention provides the ability to address these conflicts within an application program. The present invention provides a row-oriented replication. A trigger associated with a table queues deferred remote procedures. The remote procedures use the old and new values from the original modification to replicate the modification at remote sites and detect conflicting updates.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	GB	United Kingdom	MR	Mauritania
AU	Australia	GE	Georgia	MW	Malawi
BB	Barbados	GN	Guinea	NE	Niger
BE	Belgium	GR	Greece	NL	Netherlands
BF	Burkina Faso	HU	Hungary	NO	Norway
BG	Bulgaria	IE	Ireland	NZ	New Zealand
BJ	Benin	IT	Italy	PL	Poland
BR	Brazil	JP	Japan	PT	Portugal
BY	Belarus	KE	Kenya	RO	Romania
CA	Canada	KG	Kyrgyzstan	RU	Russian Federation
CF	Central African Republic	KP	Democratic People's Republic of Korea	SD	Sudan
CG	Congo			SE	Sweden
CH	Switzerland	KR	Republic of Korea	SI	Slovenia
CI	Côte d'Ivoire	KZ	Kazakhstan	SK	Slovakia
CM	Cameroon	LI	Liechtenstein	SN	Senegal
CN	China	LK	Sri Lanka	TD	Chad
CS	Czechoslovakia	LU	Luxembourg	TG	Togo
CZ	Czech Republic	LV	Latvia	TJ	Tajikistan
DE	Germany	MC	Monaco	TT	Trinidad and Tobago
DK	Denmark	MD	Republic of Moldova	UA	Ukraine
ES	Spain	MG	Madagascar	US	United States of America
FI	Finland	ML	Mali	UZ	Uzbekistan
FR	France	MN	Mongolia	VN	Viet Nam
GA	Gabon				

METHOD AND APPARATUS FOR DATA REPLICATION

BACKGROUND OF THE INVENTION

1. FIELD OF THE INVENTION

This invention relates to the field of data replication.

2. BACKGROUND ART

Because of the ability of computerized systems at one location to communicate with other locations, computer applications are increasingly accessing data located on multiple, local and remote systems. In a read-only mode (i.e., no data modifications), multiple copies of the same data item can be located at multiple sites without raising any data integrity issues. However, as multiple users resident at multiple system locations begin to modify one or more of the copies of the same data items, data integrity becomes a critical issue.

Ideally, any user should be able to modify any copy of a data item with an ability to automatically propagate the modification to each copy of the same data item at any location. Prior art systems fall short of providing this type of peer-to-peer environment.

For example, some systems provide one "master" copy with multiple "slave" copies. Any modifications are performed on the "master," with the "slave" locations receiving a copy of the modified data after the modification is completed on the "master." Thus, a user at a "slave" location must access the "master" copy to modify a data item. This technique does not provide the

ability to update any copy, and propagate the changes performed on that copy to all other copies.

For example, Cordi et al., U.S. Patent No. 4,077,059, discloses a multi-processing system with a hierarchical memory having journaling and copyback. The hierarchical memory system has two memory units on each level. One main unit contains all of the data for the level, and the other unit, the copyback unit, contains the changes that have been made to that data either by addition or modification. The main unit interfaces with the next higher level in the hierarchy and its processing unit. The copyback unit transfers the data changes to the lower level when the lower level's main unit is not interfacing with its next higher level or processing unit. The copyback unit is smaller than the main unit to reduce the necessary storage units on each level. Since the copyback unit is smaller than the main unit, there is a possibility that the number of changes to the main unit's data may exceed the capacity of the copyback's store. A monitoring routine is used to assure that the number of changes did not exceed storage capacity of the copyback unit. Appropriate measures are taken to reduce the number of changes in the copyback store when the copyback store reaches capacity.

Schmidt et al., U.S. Patent No. 4,558,413, discloses a management system for managing versions of software updated and stored in designated storage devices in a distributed software environment of a local area network. The system collects and recompiles versions of a software object (i.e., source and object modules) located on these storage devices in the LAN. The compiled program is used in the distributed software environment. The system includes the associated software object's: version, unique name, update chronology, dependencies on (and interconnections with) other software

objects, and residence. The management system is automatically notified when changes are being made to a software object.

Haas et al., U.S. Patent No. 4,631,673, discloses a method for refreshing multicolumn tables in a relational database. Haas provides a method for refreshing the snapshot (i.e., a read-only copy of a base table portion). Each record of a base table must have: (1) a tuple (i.e., unique) identifier , TID, (2) the previous records TID, PREVTID, and (3) "time stamp," UID, of the records last alteration. The snapshot contains the TID of the corresponding base table record. At refresh, the snapshot site sends the highest UID seen by the snapshot. The base table site identifies alterations based on TID, PREVTID, and UID values. Haas describes a master-slave environment whereby changes to a master are propagated to the replicas.

Kendall, U.S. Patent No. 4,635,189, describes a real-time distributed database management system that stores in a processor's memory copies of the variables needed to run the programs in that processor. When a variable is created, a processor is designated as the processor that determines the value of that variable. Each variable copy is updated by the current value of the original value on a periodic basis or upon the occurrence of a defined condition. Kendall describes a method of data manipulation such that a first processor can address an original variable in a second processor, and direct the most current value of that variable be stored in a third processor upon the occurrence of a condition in a fourth processor. An acknowledgment message can then be sent to a fifth processor.

Boyle, U.S. Patent No. 4,646,229, describes a database system that includes future versions of the database for use in time-oriented applications such as an

application for scheduling the use of the same facilities to present and future users. All of the information required to represent the data base contents at desired future points in time is maintained in the data base. All transactions (e.g., logical units of work) are time stamped to assure access to the proper version of the database.

Gallant, U.S. Patent No. 4,648,036, refers to a method for controlling query and update processing in a database system. Specifically, Gallant describes a method for ensuring that a query receives information representative of the database either before or after an update, but not information representative of a state after the update begins but before it completes. Transactional modifications are made to a future database structure. At the completion of the modifications, a switch is made from the present database structure to the future database structure. A query process accesses the present database structure.

Gladney et al., U.S. Patent No. 4,714,992, refers to a method for managing obsolescence of replicas of data objects in a distributed processing system. Database objects at a source location are replicated at a replica location. As objects stored at the source location are altered, corresponding objects at the replica location become obsolete. A replica location generates a request for a list of obsolete objects from the source location. Gladney describes a means for identifying the obsolete objects, communicating the identify of the obsolete objects, and removing the obsolete objects from the replica location. Gladney describes a master-slave environment whereby changes to a master are propagated to the replicas.

Ecklund et al., U.S. Patent No. 4,853,843, describes a multi-version database where each update operation creates a new version of the database, and the older versions remain available. Multiple alternative version paths are retained. Ecklund describes a method for deriving a minimal set of alternative version paths. When updates are applied to a partition they are performed synchronously on multiple sites in the partitions. Change list derived from the database histories and virtual partition change histories are used to determine the existence of conflicts. Ecklund describes a system for merging multiple versions of a data object into a distributed database such that each updating partition can access its own version.

Carey et al., U.S. Patent No. 4,875,159, describes a system for synchronizing two versions of files in a multiprocessor system. Both versions contain a sync-complete control field and a sync-in-progress field. The sync-complete field indicates that the associated version is synchronized when it is set. The sync-in-progress field indicates that the associated version is in the process of being synchronized when it is set. If the sync-complete field is cleared in one or both of the versions, the sync-in-progress field is set in one of the versions. Then, a temporary file is created, and a copy of the one version is transferred to the temporary filed. The sync-in-progress field of the one version is examined after the transfer is complete. If the sync-in-progress field is set, the sync-complete field is set in the temporary version. The temporary version is renamed to the other of the versions and the original of this version is removed.

Boykin et al., EPO 0,428, 264 A2, discloses a method for generating an access plan, in a database system, containing low-level code for performing preselected constraint checks. Database access commands are compiled into

access plans that are executed at runtime instead of the access commands to improve system performance.

Roussopoulos, N. and Kang, H., "Principles and Techniques in the Design of ADMS±," IEEE, December 1986, pp. 19-25 describes a technique for downloading database objects from a mainframe to a workstation as the workstation accesses data on the mainframe. Access to the localized database subset are performed at the workstation. Database objects accessed by multiple workstations are globalized. Updates at the mainframe are logged and incrementally applied before a query of the involved data is performed. Modifications to the downloaded objects are performed by maintaining a backlog consisting of entries each of which consists of an operation (i.e., insert or delete), tuple-id, and DI-log (i.e., either a pointer to a file containing deletes, or a pointer to a newly inserted tuple).

SUMMARY OF THE INVENTION

The present invention provides the ability to replicate modifications (e.g., insert, delete, or update) made at a local site to multiple remote sites in a peer-to-peer environment. Information regarding these modifications are contained in a set of replication tables. Thus, modifications can be duplicated, asynchronously, at other sites immediately after the original transaction's modifying commitment, or deferred until the remote site is available.

The replication tables of the present invention include a transactions table, transaction nodes table, calls table, call nodes table, and an exceptions table. The transactions table contains information about a transaction. The transaction nodes table contains an entry for each remote site at which a transaction is to be applied. The calls table contains an entry for each procedure (i.e., transactional or non-transactional series of logical steps) to be replicated at remote sites. The call nodes table identifies nodes at which a procedure is executed. The exceptions table contains information regarding an exception raised when processing a replicated procedure at a data site (e.g., conflict or informational message).

The present invention provides a value-oriented row level and column level replication. Row-level replication modifies a remote site based on the old and new values contained in a row at an originating site. Column-level replication modifies a remote site based on the column values at an originating site. Further, row-level and column-level replication provides the ability to identify conflicting modifications at the remote site. A trigger (i.e., a procedure) is executed each time a modification is made to a data item. A trigger queues entries in the replication tables. The table entries retain

information such that the original modification(s) associated with a data item can be propagated to remote copies of the same data items. The new values can be compared to the values contained in the remote copies to identify any modifications that could be lost, if the current modification is applied.

The present invention further provides a logic-oriented procedure-level replication. Procedure-level replication modifies a remote site based on the logical operations used to modify data at the originating site. Procedure-level replication provides the ability to identify conflicting updates as well. Thus, the same logical operations performed on one copy of a data item can be propagated to all other copies of the same data item.

Information concerning conflicts identified by the present invention can be retained in the replication tables. The information contained in the replication tables can be used immediately, or subsequently, to address any conflicts detected by the present invention. Procedure-level replication provides the ability to address conflicts, and other exceptions, within the procedure replicated to, and executed at, the remote data locations. The present invention further provides the ability to rollback any modifications made when a conflict is identified.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates one or more computerized systems at one or more locations each containing copies of data items.

Figure 2A illustrates the structure of the data items stored at one location.

Figures 2B and 2C illustrate an order transaction and sample data updated by the order transaction.

Figure 2D illustrates a replication of the transactions processed at multiple data locations.

Figure 3 illustrates replication tables.

Figure 4 illustrates a process flow of a trigger.

Figure 5A-5C illustrates a process flow for modification operations on local and remote copies of data items.

Figures 6 and 7 illustrate a technique for entering replication information in the replication tables.

Figures 8A-8B illustrate replication tables including row-level replication information.

Figures 9A-9b illustrate procedures using procedure-level replication.

Figure 10 illustrates replication tables including procedure-level replication information.

Figures 11A-11B illustrate the execution of transactional and non-transactional deferred remote procedure calls.

Figure 12 illustrates a Exceptions process flow.

Figure 13 illustrates a ReconstructParms process flow.

DETAILED DESCRIPTION OF THE INVENTION

A method and apparatus for data replication is described. In the following description, numerous specific details are set forth in order to provide a more thorough description of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without these specific details. In other instances, well-known features have not been described in detail so as not to obscure the invention.

In a networked environment consisting of one or more locations (e.g., database servers or computer sites), duplicate copies of the same data may be resident at more than one location (e.g., one or more database servers or computer systems). Figure 1 provides an example of a networked environment containing copies data resident at multiple sites. Data site A 100 may be any type of computerized system (e.g., networked database server, mainframe system, or personal computer system). Similarly, data site B 110 can be any type of computerized system. Data site A 100 contains databaseA 120. Data site A 100 and data site B 110 are interconnected via communication link 125.

Initially, databaseB 130, located at data site B 110, contains a duplicate copy of databaseA 120. Thus, a copy of the same data item is available at both sites. That is, a user that accesses databaseA can read a data item (e.g., number of brooms in inventory) at data site A 100 while another user may access the number of brooms on hand by reading information resident at data site B 110. As long as user A and B access the quantity on hand data item in read-only mode, the value of this data item at both locations will remain constant and, therefore, consistent.

Figure 2A further illustrates databaseA 120. Database A 120 contains two relations, or tables. The inventory table 202A contains two fields: Item 204A and QuantityOnHand (qoh) 204B. The item and qoh fields comprise the information for each inventory item contained in inventory table 202A. The orders table 202B contains order information associated with a customer's order (e.g., originating location, customer placing the order, item ordered, quantity of item ordered, and whether the order can be filled by stock on hand). Each table contains entries, or rows. For example, inventory table 202A contains three entries 206A-206C.

Referring to Figure 2B, DatabaseB 130, like DatabaseA, contains an inventory and orders tables. Further, DatabaseA and DatabaseB contain identical entries and values for each entry. Figure 2B further provides a set of steps that can be used to modify the data contained in either DatabaseA or DatabaseB.

This set of steps, a typical order transaction (i.e., order placement) consists of steps for checking the qoh to determine the amount of the ordered item in stock, updating the qoh in the inventory table (i.e., where qoh > quantity ordered), and inserting an entry in the orders table to reflect the order. This process is reflected in steps one and two of the basic order transaction provided in Figure 2B.

The final step in the basic order transaction, the commit step is an implicit step in transaction processing. It provides the ability to make any changes that have been made to the tables permanent. Prior to executing a

commit (i.e., makes the table changes permanent), the changes made to the inventory and orders tables can be rolled back.

Figure 2B illustrates the initial state of the tables in the two databases. However, users A and B can update the tables using the basic order transaction. That is, when user A receives an order from a customer (e.g., customer 10 orders fifty widgets) and invokes the order transaction at location A (i.e., databaseA), the order transaction will update the inventory and orders tables in DatabaseA. DatabaseB will remain unchanged. Thus, after user A's order processing transaction, DatabaseB will no longer be identical to DatabaseA.

Figure 2C illustrates the resulting databases after both user A and B use the order transaction to process an order received at their respective sites. For example, user A receives an order for fifty widgets from customer 10. User A invokes the order transaction. The order transaction updates the inventory table (i.e., the qoh field associated with the widget inventory item) and the orders table (i.e., adds an entry to reflect the order) in database A. Similarly, user B invokes the order transaction to process the order received from customer 11 at site B (i.e., for forty widgets). The transaction invoked by B updates the inventory table and orders table in database B.

Both orders consisted of an order for widgets. Customer 10's order was for fifty widgets, and Customer 11's order was for forty widgets. Therefore, a total of ninety widgets were sold. However, database A does not reflect customer 11's order, and database B does not reflect customer 10's order. Thus, each database only reflects the orders processed by one of the users (i.e., A or B), and will therefore not reflect all of the orders that have been processed at all of the sites.

Thus, there is a need to propagate local modification to all remote copies of the same data item. The present invention provides this ability to replicate the data modifications made at one location to other locations. Thus, the databases illustrated in Figure 2C are be replicated to other sites such that order transaction modifications at one site are be applied to other sites, and the data items are again consistent.

Figure 2D illustrates the state of the two databases before and after the replication capabilities of the present invention are used. Before replication, the two databases reflect only the order transactions processed locally (i.e., database A reflects user A's order transaction, and database B reflects user B's order transaction), and not the order transactions processed on a remote database (e.g., database B is a remote database to user A).

After the DbB=>DbA and DbA=>DbB replications, user A's data modifications are reflected at user A's local database (i.e., database A) and at database B (i.e., remote site). Similarly, user B's data modifications are reflected at both the local and remote sites. The widget's qoh value in both of the databases reflects the overall decrease in the qoh associated with both of the orders. Further, the orders tables in both databases reflect both of the orders received.

The present invention provides the ability to replicate data at the row level and at the procedure level. Row level replication (i.e., value-oriented replication) is accomplished by associating a trigger with a table (e.g., inventory table). A trigger is a procedure that is executed when a modification (e.g., update, insert or delete) occurs to a row in a table. A trigger identifies a

deferred remote procedure call (DRPC) that has as its arguments the old values, new values, and the operation (e.g., insert, delete, or update).

Procedure-level replication propagates the operation rather than the row values (i.e., logic-oriented). After a procedure is executed at the originating site, the procedure defers a call to itself at another site. The DRPC will apply the logical update of the original procedure at a remote site. Procedure-level replication requires less network traffic than row-level replication since one DRPC can be used for multiple tables.

Modification, Identification and Retention

To propagate the modifications made to data items in one database to the same data items in another database, it is necessary to retain the modifications until they can be made to the other sites. Prior art methods use a transactional recovery log (i.e., redo log) to retain and identify the database modifications for propagation. However, the redo log was originally intended for the application of transactions in a single database system, and contains "undo" information (i.e., information that can be used to roll back changes made to data after an event, such as a system failure, occurs).

For example, a redo log can be used on a single copy of the data to undo changes made to a database by one or more transactions when a system or application error is encountered. When such an error is encountered, the transactional updates made prior to the error (and related in some way to the error), must be undone to maintain the data integrity in existence prior to the updates. However, the redo log was not designed for capturing modification information for propagation to a second database system. Thus, when the log

is used for its intended purpose as well as a resource for data replication, a storage management problem arises, because a redo log that it used to retain propagation information can never be moved off-line (i.e., made inaccessible to the database system).

Unlike the prior art systems, the present invention provides a propagation identification capability that can be managed in the same way as any other table, or relation, managed by a database administrator. The present invention provides the ability to encode the propagation information in tables in the database system. The information stored in these tables can be retrieved as any other data within the database system, and can be accessed at any time.

The tables include the information necessary to replicate a data modification to other data sites. For example, the tables contain information relative to a DRPC , its replication destination, the transaction the DRPC is a part of, the order in which a DRPC is executed within a deferring transaction, the order in which transactions are executed relative to other transactions, and the arguments used by each DRPC. The present invention uses the following tables: Transactions, Transaction Nodes, Calls, Call-nodes, and Exceptions. Figure 3 illustrates a composition of these tables. Because of the versatility of the present invention (e.g., replication information is stored in relations), additional information can be added to these relations.

-- Transactions Table --

The Transaction Table contains information about transactions that are performed on the data and that use deferred remote procedure calls (i.e., DRPCs) in some manner. The transactions table consists of the following

fields: transaction identifier, delivery order number (DON), start time, deferring user identifier, and destination-list. The Transaction identifier ("Transaction_Id") is a unique identifier that is assigned to each transaction. The Transaction_Id further uniquely identifies the origin database for deferred transactions. Transaction_Id is the primary key for the Transaction table (i.e., the value that uniquely identifies an entry in this relation). The destination-list controls whether destinations for a transaction are described by the call-nodes table or external routing tables.

The DON is an abstraction of a system change number. A DON reflects the commit sequence of a transaction relative to all of the other transactions listed in the transaction table. The DON respects the partial order of the commit of the transactions in the deferring database. Thus, if transaction one T1 touches the same data as transaction two T2 and T2 commits before T1, the DON of transaction two (D2) is less than T1's DON (D1).

The time field reflects the time that the transaction was started, and the deferring user field identifies the user who initiated the deferred transaction's procedure call. This information can be used to monitor and control access (i.e., for security reasons). For example, prior to making any modifications to data at a remote site, a check can be made to determine whether the deferring user has the access privileges that would enable the modification at the remote site.

The replication capability provided by the present provides the ability to modify all or some portion of the data modified by a transaction by site specification. That is, a deferred transaction can consist of a subset of the calls deferred by the original transaction for any given destination site. For

example, an order transaction at site A can update an orders and inventory tables, and the replicated transaction at site B can update the orders table only.

A call nodes table can be used to define a transaction's calls that are applied at a given site. In addition, any routing mechanism can be used. For example, a mapping of calls to destinations can be defined according to the name of the procedures beings deferred. The destination list field of the transactions tables indicates the mapping mechanism (e.g., call nodes table or other routing mechanism) used.

– Transaction Nodes Table –

The Transaction Nodes Table (i.e., destinations table) identifies the nodes or remote sites at which the transactions contained in the transactions table are to be executed. The transaction node table contains one entry for each node (i.e., remote site) at which a transaction is to be executed.

The transaction identifier ("Transaction_Id") has the same definition as the same field in the transactions table. The Destination Node ("dest_node") identifies the nodes (i.e., remote databases) at which the transaction is to be executed to replicate the changes made by the transaction on the local data site. Thus, the same transaction_id can be used to access an entry in the transaction table, and any corresponding entry or entries in the transaction node table. Further, an entry in the transaction nodes table identifies one of the remote sites to which the transaction's modification are to be propagated. A transaction_id and dest_node combination can uniquely identifies an entry in the transaction nodes table.

-- Calls Table --

As illustrated in the order processing example provided earlier, transactions (i.e., a logical unit of work) can be composed of steps, or procedures (e.g., place order and inventory check). In software encoding these steps, each can be considered a separate procedure to provide additional structure to an application. Further, procedures can be defined without being a part of a transaction. Information regarding either type of procedure is retained in the calls table. The calls table contains a unique identifier, Call Identifier ("call_id"), that can order a call within a transaction, or orders non-transactional DRPCs relative to all others.

Like the transactions and transaction nodes tables, the calls table contains a transaction_id. For transactional DRPCs, the transaction-id has the same definition as the transaction_id field in the transactions and transaction nodes tables. For non-transactional DRPCs, the transaction_id field is not used.

The Deferred Procedure Identifier ("proc_id") identifies the procedure, or call, (i.e., series of program steps) that is to be executed at the remote site. For example, the proc_id may be a character string containing the procedure's name. It could also be a system-provided, unique identifier that includes the location (e.g., storage address) of the procedure. The parameter count identifies the number of parameters (values passed into the procedure for use during the execution of the procedure) for a procedure.

The parameters field (parms) is a long raw byte string containing the parameters for the entry in the calls table. The format of the field is as follows:

<tc1><len1><value1><tc2><len2><value2> . . . <tcn><lenn><valuen><0>

Where:

<tci> is the parameter type code for the ith parameter (i.e., whether the parameter is of type number, character, date, rowid, or null);

<leni> is the two byte binary integer value of the length of valuei (length of zero indicates a null parameter value);

<valuei> is the parameter value;

<0> is a single byte value indicating the end of the string.

– Call-Nodes Table –

The call-nodes table contains a row for every destination of each deferred call when the destination is not defined by an external routing table. The call-nodes facilitates the ability to replicate a transaction comprised of multiple procedure calls by specifying the execution of some or all of the calls at a given site. When a call's destinations are not defined by an external routing structure, call nodes are specified by the deferring user, either with the deferred call, as transaction default destinations, or as system determined destinations. When a deferred transaction is being sent to a destination, the call-nodes table is queried to select those calls that are to be executed as part of the deferred transaction at the destination.

The Transaction Identifier and the call identifier are the same as those in the calls table. The destination node field identifies the node at which the execution of a procedure is deferred.

-- Parameters Table --

In an alternate embodiment, a table can be used to retain the parameters for a call instead of storing the parameters in the calls table. In this embodiment, the parameters table contains an entry for each parameter used by an entry in the calls table. That is, for each call containing parameters, there is one or more entries in the parameters table. Each entry in the parameters table contains a parameter for an entry in the calls table.

The parameters table contains a Call Identifier ("call_id"). Like the calls table, the call-id identifies a procedure call. A procedure call with more than one parameter contains an ordered list of parameters. The Parameter Number ("param_no") can, therefore, identify an entry's location within a procedure call's ordered list of parameters. A call_id and param_no pair can uniquely identify an entry in the parameters table. The type field contains a code that indicates the type of parameter. That is, the type field indicates whether the parameter table entry is a number, character, date, rowid.

Only one of the remaining fields (i.e., Number, Character, Data, Rowid) is used for each entry in the table. That is, if the parameter is a number, the value of the parameter is stored in the number field. Similarly, if the parameter is of type character, the value of the parameter is stored in the character field. A date parameter value is stored in the date field. Rowid

information (i.e., identifier specifying a row within a table) is stored in the rowid field.

-- Exceptions Table --

The exceptions table is used to store information related to any exceptional or occurrence during executions of a deferred transaction. This information can be subsequently reviewed, and the exceptional occurrence can be addressed. For example, multiple conflicting updates may occur to different copies of replicated data. Thus, one transaction, T1, can update one copy of record A, C1, and a second transaction, T2, can update a second copy of record A, C2. If T1 is propagated to C2, T1 can overwrite T2's update, and vice versa. The present invention detects this type of exception, and others, and retains information for each exception.

The exceptions table contains a Transaction Identifier ("transaction_id") field that has the same definition as the transaction_id field in the transactions, transaction nodes, and calls tables. The call identifier has the same definition as the call_id field in the calls table. The Destination Node ("dest_node") identifies the node at which the exception occurred. In the previous paragraph's example, the node field would contain the identification of the node that stores C2. The error code field (error_code) contains a code to identify the error, or exception, encountered (e.g., overwrite potential for T2's update of C2). Further, the error string field contains an additional description of the error. A transaction_id, and dest_node combination can uniquely identify an entry in this table.

Populating Replication Tables

The modification replication provided by the present invention is asynchronous. That is, the replication procedures that modify the remote data copies (e.g., <table_name>_insert) do not execute as part of the modification operation that is performed at the local, originating site. Rather, a modification to the remote data copies can be deferred until the remote copies are available.

The process of deferring the modification operation at the remote site is accomplished, in the present invention, by storing the information for each deferred modification operation in the replication tables, and subsequently performing the modification operation identified in the replication tables at the remote sites.

Referring to DatabaseA (DbA) in Figure 2C, for example, the modifications to the Inventory and Orders tables can be replicated to DatabaseB (DbB) in Figure 2C by replicating DbA's modification to DbB's Inventory and Orders tables. Thus, the update performed on DbA's inventory table and the entry inserted in DbA's orders table can be replicated at the DbB site by replicating the Basic Order Transaction's modifications performed on the data in DbA.

The basic order transaction is replicated at DbB by queuing the transaction in the replication tables, and subsequently applying the modifications contained in the transaction on the data in DbB according to the information contained in the replication tables. Figure 6 illustrates a processing flow for queuing the transaction, transaction nodes, and call

destination tables for a transaction, a transaction's destination nodes, and any transactional or non-transactional call. Figure 6 can be invoked multiple times to queue any number of calls. For example, the invoking routine can contain a looping mechanism that can invoke queue_transactional_DRPC for each transactional or non-transactional call.

At decision block 601 (i.e., "first call?", if this is not the first call to queue this transaction, processing continues at block 616. If it is the first call, processing continues at block 602. At processing block 602, a current transaction identifier is assigned to the transaction to be stored in the tables (e.g., DbA's basic order transaction). The current transaction identifier is assigned a value that will uniquely identify the table entry. At processing block 604, an entry is inserted in the transactions table. The transaction identifier field is assigned the value of the current transaction identifier.

The originating transaction (e.g., DbA's order transaction) is assigned a DON when the commit step in the order transaction is successfully performed. The DON provides an ability to order transactions based on the order in which they have modified the data. Thus, where the order of the modifications is crucial, the DON can be used to retain the modification order and thereby maintain data integrity. The DON field is assigned the value of the original transaction's DON. The time field of the new transactions table entry is set to the current time. The deferring user identifier is assigned the value of the user that originated the original transaction.

A destination (i.e., remote data copy identifier) is identified at processing block 606. At processing block 608, an entry in the transaction nodes table is created to indicate that the transaction currently being entered into the

replication table is to be performed at the identified destination. Thus, the transaction identifier is assigned the same value as the same field in the transactions table, and the destination node is set to the identified destination.

An entry should be made for each destination identified. Thus, at decision block 610 (i.e., "other remote destinations?"), if additional destinations exist, processing continues at processing block 606 to identify the next destination. Further, a transactions node table entry is created for each such destination. When it is determined, at decision block 610 (i.e., "other remote destinations?"), that all of the remote destinations have been entered into the transactions node table, processing continues at decision block 616 (i.e. "all calls in transaction processed?")

As illustrated in the basic order transaction provided in Figure 2B, a transaction can be comprised of multiple steps. In the basic order transaction, the steps where: check inventory and process order. These steps can be designed and encoded in software as separate procedures. In this case, the basic order transaction can contain an inventory_check procedure that exercises the steps contained in an inventory check. Similarly, the basic order transaction can contain the place_order and commit procedures. Each of these calls that comprise the transaction can then be entered into the calls table of the present invention.

At processing block 616, a unique call identifier is generated and assigned to current_call_id. At processing block 618, Queue_call_args is invoked to enter the call information into the replication tables. After the appropriate call information has been added to the replication tables, processing returns at block 620.

Figure 7 illustrates a processing flow to queue call information. At processing block 700, an entry is created in the calls table for the procedure call currently being processed. The entry's call identifier field is set to the current_call_id. The same value assigned to the transaction identifier fields in the transactions and transaction nodes tables is used for the transaction identifier field in the calls table. The procedure identifier field is assigned a value that can uniquely identify the procedure being deferred. This value can be anything that identifies the procedure such as a procedure name or the storage location of the procedure.

During its execution, a procedure can use values (parameters) that are externally-generated (i.e., defined outside the procedure) and passed into the procedure, or internally-generated. In the preferred embodiment, the parameters are queued in the parms field of the calls table.

At processing block 702, parm_count is initialized to zero. At decision block 704 (i.e., "all parameters in call processed?"), if all of a procedure's parameters have been added to the parameters table, or the procedure does not have any associated parameters, processing returns at processing block 714. At block 714, the value of parm_count is used to update the parameter count field of the call's entry in the calls table.

At decision block 716 (i.e., "call's destination nodes not defined by external routing mechanism and destination nodes defined by user with DRPC?"), if the call's destination nodes (i.e., execution nodes) are defined by an external routing mechanism, processing returns at block 724. If the call's execution nodes are defined by the user with the DRPC and not by an external

routing mechanism, processing continues at decision block 718. At decision block 718 (i.e., "all execution nodes processed?"), if all of the execution nodes have been entered in the call nodes table, processing returns at block 724.

If all of the execution nodes have not been processed, processing continues at block 720 to get the next execution node specified by the user. At processing block 722, an entry is created in the call nodes table for the current execution node. Processing continues at decision block 718 to process any remaining execution nodes.

If, at decision block 704 (i.e., "all parameters in call processed?"), parameters remain to be processed, processing continues at processing block 710. At processing block 710, the parameter's type (i.e., data type), length, and value are appended to any existing value in the parms field of the current calls table entry. At block 712, parm_count is incremented by one. Processing continues at decision block 704 to process any remaining parameters.

In an alternate embodiment, a call's parameters can be stored in a separate table, a parameters table (see Figure 3). In the alternate embodiment, block 710 creates a separate entry in a parameters table for each parameter. Thus, a parameters table contains an entry for each parameter that is associated with a procedure. Each entry contains the call identifier, a parameter number (i.e., the number of the parameter in relation to the other parameters in the call), and the data type of the parameter. The value of the parameter is stored in one of the value fields (i.e., number, character, date, or rowid) based on the data type. For example, if the inventory_check procedure in the basic order transaction contained three parameters, three entries would be added to the parameters table.

Triggers

Triggers provide one alternative to initiate the population of the replication tables . A trigger is a procedure that is executed when any modification (e.g., update, insert or delete) is performed on an entry in a table at the local site. Figure 4 provides an example of the process flow of a trigger in the present invention.

Decision block 402 (i.e., "is this trigger firing as a result of a replicated modification?") illustrates an issue addressed by a trigger in the present invention. Because a trigger is initiated when any modification operation is performed on a table, an operation that is performed on a remote data entry (i.e., a modification operation) will result in the initiation of a second trigger.

Unless a trigger contains some mechanism for differentiating between a modification done as a result of an original modification operation and a modification that is a replicated modification (i.e., the result of an earlier-fired trigger), the replicated modification will itself generate a trigger, and the original modification operation could be replicated multiple times at a data site unnecessarily. This would endanger the integrity of the data at the local and remote sites.

Thus, decision block 402 (i.e., "is this trigger a result of an earlier-fired trigger?") determines whether the table modification that generated the trigger is a consequence of an original or replication modification. If it is an original modification, the trigger is not the result of an earlier-fired trigger. However if the modification is the result of an original modification that has been

replicated to a remote table, a second trigger should not be queued in the replication tables.

Therefore, if, at decision block 402 (i.e., "is this trigger a result of an earlier-fired trigger?"), a trigger is the result of replicated modification, a processing ends at block 406, and the modification procedure is not queued. If, at decision block 402, the trigger was generated as a result of an original modification, processing continues at processing block 404. At processing block 404, a modification entry is inserted in the replication tables. Processing then ends at block 406.

The mechanism for identifying duplicative replications can be implemented in various ways in the present invention. In the preferred embodiment, a duplicative replication can also be detected by setting a global variable (i.e., a variable that can be accessed by any trigger or replication procedure) before a replicated modification is performed. When the replicated modification is performed, the trigger can (at decision block 402 in Figure 4) check the global variable to determine whether the modification is the result of a replication procedure or an original modification. This alternative of setting the global variable in the replication procedure is further illustrated in connection with row-level replication discussed below.

In an alternate embodiment, a duplicative replication can be detected by associating data modifications to a user. Thus, it is possible to identify an original modification by its user, and to identify a replicated modification by another user (i.e., modification procedures can be executed by a distinguished user). Thus, decision block 402 in Figure 4 can check for the user name that invoked the modification. If the user is a distinguished user, the trigger was

generated as a result of an earlier-fired trigger. If the user is not a distinguished user, the trigger was generated as a result of an original modification.

These alternatives provide examples of techniques for detecting duplicative modifications. Any means can be used to detect duplicative modifications without departing from the scope of the present invention.

Row-Level Replication Procedures

Row-level replication is a feature of the present invention that uses triggers to replicate row-level value-oriented modifications. That is, row-level replication provides an ability to replicate changes made to the values in a row. Row-level replication associates a trigger with a table (e.g., DbA's inventory in Figure 2A) such that any changes made to one or more values in a local table entry (e.g., qoh field in DbA's inventory table of Figure 2A) will trigger a like change to remote copies of the changed values (e.g., qoh field in DbB's inventory table of Figure 2A).

A trigger causes the information associated with a procedure used to replicate the value changes in the local copy to a remote copy to be stored in the replication tables. The procedure, a deferred remote procedure call (DRPC), can be subsequently executed at remote sites to replicate the data modification(s) performed on local data. The name of the DRPC corresponds to the table being modified and the operation being performed on the local data (e.g., <table_name>_update). The DRPC has as its arguments (i.e., parameters), generally, the old values of the local data and the new values of the local data. The old values, or a subset thereof, uniquely identify the row

that is the target of the modification. The use of these arguments is specific to the operation that is performed at the local site.

For example, if an update operation is performed on the local data, the old values would be used to detect conflicts. That is, a difference between the old values of the local data and the current values at the remote site may indicate that a separate operation has been performed on the remote data that may get erased with the current update. The new values can be used to update the remote data.

An insert operation (i.e., insert a row of values or a field value) there are no old values, and, therefore, no old values are included in the call. Further, there is no need to use old values to perform a check for conflicts (i.e., exceptions) at the row-level when inserting a new row or field value. The new values, or a subset thereof, uniquely identify a new remote table entry.

If the operation is a delete (i.e., delete a row or field), there are no new values. However, like an update operation, the old values can be used to detect potential conflicts. Further, the old values, or a subset thereof, can be used to uniquely identify the remote table entry to be deleted.

A DRPC name can incorporate the name of the table to be modified and the operation to be performed. Referring to Figure 2A, for example, the triggers for the inventory table can have the names: inventory_insert, inventory_update, and inventory_delete). This naming convention assists in identifying the table and operation involved in the replication process. However, any naming convention can be used with the present invention.

– Row-level insert –

Figures 5A-5C provide an example of the process flow for the <table_name>_insert, <table_name>_delete, and <table_name>_update DRPCs. These DRPCs execute at sites remote from the original modification. Therefore, when they refer to "remote tables," they are referring to tables local to the DRPC's execution and remote from the original modification. Figure 4A provides an example of the processing flow of a <table_name>_insert DRPC. As discussed previously, a global variable can be used as one alternative for identifying replicated modifications. Such a global variable is set at processing block 502. At processing block 504, a row is inserted in <table_name> using the value(s) provided in the new_values parameter(s).

To further illustrate the need to test for duplicative modifications, the replicated insert operation performed by the process of Figure 5A would generate a trigger (i.e., any table modification initiates a trigger). Thus, the trigger process of Figure 4 is invoked when the insert operation of processing block 504 (Figure 5A) is performed. Because the global variable was set to indicate a duplicative modification (at processing block 502 of Figure 5A), the trigger can determine (at processing block 402 of Figure 4) that the modification is a replicated modification, and a DRPC will not be queued for the replicated modification (i.e., at processing block 404 in Figure 4).

Continuing with the process flow of Figure 5A, after the insert operation is performed on the remote table, the global replication variable is reset at processing block 506. Processing ends at block 508.

– Row-level update –

Figure 5B provides an example of the processing flow of an update DRPC (e.g., <table_name>update). A global replication variable is set at processing block 522. At processing block 524, the remote table entry is identified using the old values, or a subset of the old values. At decision block 526 (i.e., "row found at remote site?"), if the remote table entry cannot be found, processing continues at processing block 532 to log an exception in the replication tables. Further, the global replication variable is reset at processing block 548, and processing ends at block 550. If the remote table entry is found, processing continues at decision block 530.

Before an update is made to a remote site, a check can be made to determine whether a modification has been made to the remote data that is independent of the current update operation that might be erased if the current update is performed on the remote data. This might occur, for example, when a modification (other than the current, replicated operation) could have originated at the remote site, the replication of which has not reached the site that invoked the current replicated update. If the current replicated update overwrites the remote table entry's values with the new-value parameters, the remote table entry's current values will be lost, and the remote table's original modification will, therefore, be lost.

Alternatively, in some applications, concurrent modifications to disjoint sets of non-primary field values can be permitted. For example, a modification to a customer's balance need not conflict with a change to a customer's address. If updates can be applied to non-primary fields on a field-by-field basis, concurrent updates are not lost. At decision block 530 (i.e., "type

of lost update prevention?") determines whether the lost update is a row level update or a column level update. If it is a row level update, processing continues at decision block 536. If it is a column level update processing continues at decision block 534.

At decision block 534 (i.e., "is each field value equal to its corresponding old_value parameter where the corresponding old value parameter is not equal to the corresponding new value parameter?), if the old values are equal to their corresponding old_value parameter where the old value parameter is not equal to the new value parameter, processing continues at processing block 544 to update the fields that have been changed, and processing continues at block 548. If not, processing continues at decision block 538 (i.e., "should lost updates be prevented?"), if lost updates should be prevented, processing continues at block 540 to invoke Exceptions, and processing continues at block 548.

At decision block 536 (i.e., "is each field value equal in the row equal to its corresponding old_value parameter?), if the old values are equal to their corresponding old_value parameter, processing continues at processing block 546 to update each field in the row with its corresponding new_value parameter, and processing continues at block 548. If not, processing continues at decision block 542 (i.e., "should lost updates be prevented?"), if lost updates should be prevented, processing continues at block 540 to invoke Exceptions, and processing continues at block 548.

At processing block the global replication variable is reset. Processing ends at block 550.

-- Row-level delete --

Figure 5C provides an example of the processing flow of a delete DRPC (e.g., <table_name>delete). A global replication variable is set at processing block 562. At processing block 564, the remote table entry is identified using the old values, or a subset of the old values. At decision block 566 (i.e., "row found at remote site?"), if the remote table entry cannot be found, processing continues at processing block 572 to log an exception in the replication tables. Further, the global replication variable is reset at processing block 576, and processing ends at block 578. If the remote table entry is found, processing continues at decision block 528.

As in the update DRPC process, a check is made to determine if a check should be made for lost updates (i.e., modifications). Thus, decision block 568 (i.e., "should lost updates be prevented?") determines whether to test for potential lost updates (i.e., lost updates). If not, processing continues at processing block 574, and the remote table entry is deleted from the remote table. After the delete operation is performed on the remote table entry, the global replication variable is reset at processing block 576, and processing ends at block 578.

If, at decision block 568 (i.e., "should lost updates be prevented?"), existing modifications should be preserved, processing continues at decision block 570. At decision block 530 (i.e., "is each field value in the row equal to its corresponding old_value parameter?"), if any of the remote table entry's field values do not equal its corresponding old_value parameter, processing continues at processing block 572. At processing block 572, an exception is

logged in the replication tables. Processing continues at block 576 where the global replication variable is reset, and processing ends at block 578.

If, at decision block 570 (i.e., "is each field value in the row equal to its corresponding old_value parameter?"), all of the field values in the remote table entry are equal to their corresponding old_value parameters , processing continues at processing block 574. At processing block 574, the remote table entry is deleted from the remote table. After the delete operation is performed on the remote table, the global replication variable is reset at processing block 576, and processing ends at block 578.

Row-level Replication Example

The replication illustrated in Figure 2C (i.e., DbB=>DbA and DbA=>DbB) can be accomplished using triggers and row-level replication. The order for fifty widgets at the database A location resulted in the invocation of a basic order transaction to update the inventory table and place an order in the orders table. Either of these modifications will cause a trigger associated with either table to execute.

For example, when the qoh field of DbA's inventory table is updated by subtracting the quantity ordered, a trigger associated with the inventory table (and illustrated in Figure 4) invokes the procedures of Figures 6 and 7 to populate the replication tables with a DRPC. In this case, an Inventory_update DRPC similar to the <table_name>_update DRPC illustrated in Figure 5B can be used to replicate the changes in DbA's inventory table to DbB.

Referring to Figure 6, if this is the first call, a transactional identifier is generated for the DRPC (i.e., ,<table name>_update) at processing block 602. At processing block 604, an entry is inserted into the transactions table as illustrated in Figure 8A. A transaction identifier (e.g., 1), DON, time, and deferring user identifier are assigned to the inventory_update transaction.

At blocks 606 through 610, the transaction nodes table is populated. In this case, the only remote copy of the data is located at DbB. Therefore, one entry is inserted in the transaction nodes table where the transaction identifier is the same as the same field in the transactions table, and the destination node is set to DbB.

An entry is created in the calls table for each call in the transaction. Referring to Figure 7, an entry is inserted into the calls table to reflect the inventory_update DRPC. The call identifier is a unique identifier for the inventory_update DRPC. The transaction identifier has the same value as the same field in the transactions and transaction nodes tables. The deferred procedure identifier can be any value that identifies the DRPC. In this case, the name of the DRPC is used.

If an external routing mechanism is not being used, the call nodes table can be populated with the entries that identify the user specified destination nodes at which execution of the DRPC is deferred (i.e., execution nodes). Referring to 8A, the destination list field indicates that the destination nodes are not specified by an external routing mechanism. In this case, the destination nodes for the transaction's one DRPC (i.e., inventory_update) can be determined by the entry in the transaction nodes table. However, the call

nodes table can be used to identify DbB as the destination node for execution of the inventory_update DRPC.

Any parameters associated with a call are stored in the call's entry in the calls table. Referring to Figure 5B, the update procedure uses the widget's old inventory values and new inventory values. The widget entry in the inventory table is uniquely identified by the old value "widget." Because each entry in the table contains two fields, there will be two old values and two new values. Therefore, there are four arguments associated with the inventory_update procedure. The arguments (including an example of their attributes) are as follows:

	Old values			New values	
Type	Length	Value	Type	Length	Value
2	06	Widget	2	06	Widget
1	03	400	1	03	350

The parms field in the calls table entry associated with this call contains a string of parameter information. A terminating value (e.g., "0") is placed at the end of the string. The resulting string is:

"206Widget103400206Widget1033500".

If inventory_update is considered to be a non-transactional DRPC, the process of inserting entries in the transactions and transaction nodes tables could be bypassed. To associate the non-transactional DRPC with the destinations at which the DRPC is to be executed, the call nodes table can be used. An entry can be placed in the call nodes table for each location at which the call is to be executed.

Figure 8B illustrates the state of the replication tables after their population with a non-transactional DRPC. The transactions and transaction nodes tables are not used. The calls and call nodes tables are the same as in Figure 8A with the exception of the transaction identifier field. Since there is no associated transaction, there is no entry in the transaction and transaction nodes tables, and no transaction identifier value.

Column-Level Replication

Column-level replication is a variation of lost update prevention for row-level replication. Column-level replication applies only to update operations. In column-level replication, concurrent updates can be made to disjoint sets of non-primary key columns. Lost updates are prevented only when the updates are to columns whose values have been changed (i.e., those columns changed at the originating site as indicated by a difference between the old_value and new_value parameters).

Column-level replication uses the same <table name>_update procedure as row-level replication. Figure 5B illustrates the differences in the logic for detecting lost updates and applying updates for row-level and column-level replication schemes. A row-level replication scheme determines that a remote site's current values match old values for all columns in a table entry (at decision block 536) prior to applying an update at a remote site. Column-level replication checks (at decision block 534) only those columns that were changed by the original update (as indicated by a difference between the values of the corresponding old and new parameters). If the old_value parameters of the changed columns are equal to their corresponding values at the remote site, the loss of an intermediate update is

unlikely, and the update operation can be completed. If the old_value parameters of the changed columns are not the same as their corresponding values and lost updates are to be prevented (at decision block 538), an exception is raised (processing block 540), and the update operation is not performed. If lost updates are not to be prevented (at decision block 538), the update operation can be performed.

A row-level update operation involves all of the columns in a table entry (at processing block 546). However, a column-level update operation involves only those columns changed by the original update (as indicated by a difference between the values of the corresponding old and new parameters). Thus, in a column-level update operation, only those columns changed by the original update are updated with their corresponding value in the new_value parameters (processing block 544).

Procedure-Level Replication

Procedure-level replication provides another alternative for replicating data modifications. As previously discussed, row-level replication is value oriented. That is, the values that are the result of some operation are replicated to remote copies. In contrast, procedure-level replication provides the ability to replicate the logical operation at remote sites. That is, the procedure that modified the local copy can be replicated at the remote sites. Thus, after the execution of a procedure at the originating site, the procedure creates a DRPC such that the procedure defers itself to another site to apply the logical update on a remote copy.

Procedure-level replication provides additional flexibility in that an application (e.g., order processing example in Figures 2A-2C) can determine how a replication is to be propagated, and how replication conflicts (e.g., multiple conflicting updates to the same data item) are to be addressed. That is, a DRPC can designate its propagation and the process to be invoked when replication conflicts are identified.

Figures 9A-9B provide an example of an application that uses procedure-level replication to replicate its logical operations to other sites. The application assumes the relations (i.e., tables) described in Figures 2A-2D. Further, any changes made to the tables at the DbA site must be replicated at the DbB site, and vice versa.

In the basic order transaction illustrated in Figures 2B-2C, once a customer's order is received, the inventory table is updated to reflect a decrease in inventory by the quantity ordered, and an entry is inserted in the orders table to retain information about the order. If the order is received and processed at DbA, the same order processing is replicated at DbB as illustrated in Figure 2D.

Figure 9A illustrates the original order processing that is performed when an order is received at one of the sites (e.g., DbA). To summarize the order processing at DbA, at processing block 902, the item ordered is found in the inventory table, and the inventory amount is stored in qoh. If the entry is not found, an exception is raised at block 906. If the inventory amount (i.e., qoh) is greater than the quantity ordered, the inventory amount is decreased by the quantity ordered, and the order is considered to be filled at blocks 908, 912, and 914. If the inventory amount is not greater than the quantity ordered, the

order is considered to be backordered. In either case, the order is entered into the orders table at processing block 916.

Once the order is processed at DbA, the order processing is replicated at remote sites by entering the replicated procedure in the replication tables, and subsequently executed the replicated procedure at the remote sites. At processing block 918, the information associated with the replicated procedure is stored in the replication tables by invoking Queue_transactional_DRPC. As described earlier, Queue_transactional_DRPC stores the replicated information in the tables. Processing block 918 further illustrates some of the information that is stored in the replication tables.

Figure 10 illustrates the state of the replication tables after Queue_transactional_DRPC has processed the replicated procedure's information. The transactions table is populated with a transaction identifier to uniquely identify the order processing transaction, as well as the transaction's DON, commit time, and deferring user. The transaction nodes table has one entry for the remote copy of the inventory and orders tables located in DbB.

The calls table contains an entry to identify the place_order_remote procedure for this order processing transaction (i.e., 4), and the parameter count is set to five. The parameters field in the calls table contains the parameter information (i.e., item ordered, originating site, customer, quantity ordered, and the order's status in the DbA database). The call nodes table contains an entry to identify the node at which the place_order_remote DRPC is to be executed.

Figure 9B provides an example of the remote processing of the place order process at DbB. Except for the change in the data copy being modified, the processing blocks 902 through 916 are the same as the same blocks in Figure 9A. As stated previously, procedure-level replication provides the ability to allow an application to handle exceptions that occur as a result of the replication of the procedure at other sites. In the present example, a check is made to determine whether there is a discrepancy in the ordered item's inventory count in DbA and DbB. One way of determining this is by determining whether the order could be filled based on the QuantityOnHand information in both databases. Thus, at decision block 920 in Figure 9B (i.e., "status = filled?"), the order's status at DbB is checked against the order's status at DbA. If they are not equal, an exception can be raised for later review.

Instead of raising an exception for later review, other methods of addressing this discrepancy can be included in the procedure. In the present example, the procedure could have been designed to either modify the order at DbA, or at DbB. In any case, the present invention provides the ability to allow an application to process exceptions using procedure-level replication. Whether or not a discrepancy is detected at decision block 920, processing ends at 924.

Deferred Remote Procedure Call Initiation

As previously indicated, once a DRPC has been placed in the replication tables, the present invention provides the ability to subsequently execute the DRPC at a remote site. Figure 11A illustrates a process for initiating deferred, transactional DRPCs contained in the replication tables.

The selection of transactional DRPCs can be performed using a number of different criteria. For example, they can be selected based on the transaction identifier, transaction destination, or a combination of the two. Whatever the selection criteria, the transactions to be processed are selected at processing block 1102. Select transactions are ordered for execution according to the DON field in the transaction table entries. At decision block 1104 (i.e., "all selected transactions processed?"), if all of the selected transactions have been executed, processing ends at 1132.

If there are remaining transactions, the next transaction is obtained, and its identifier is stored in current_trans_id at processing block 1106. The modification operations performed in a transaction can be undone before they are committed. At processing block 1108, a point (i.e., savepoint) is established to identify the state of the data prior to the modifications of the current transactions.

At decision block 1110 (i.e., "all calls processed?"), if all of the calls in the current transaction have been processed, processing continues at processing block 1112. At block 1112, the entry in the transaction nodes table that corresponds to the transaction processed and the remote copy modified are deleted from the transaction nodes table. At decision block 1114 (i.e., "anymore remote copies to apply DRPC to?"), if there is some need to retain the current transaction's entry in the replication tables, processing continues at processing block 1118. If there is no need to retain the entry in the replication tables, the entry is deleted from the replications table at processing block 1116. At processing block 1118, the current transaction's modifications are committed making its modifications permanent. Processing continues, at decision block

1104 (i.e., all selected transactions processed?"), to perform the remaining transactions.

If at decision block 1110 (i.e., "all calls processed?"), if all of the DRPCs have not been executed in the current transaction, processing continues at processing block 1120. At processing block 1120, a call nodes table entry for the destination node currently being processed. Processing block 1121 identifies a calls table entry for the call identified in processing block 1120.

At processing block 1122, the DRPC call string is reconstructed. A DRPC call string is one technique for identifying the DRPC to be executed. Other means for identifying the DRPC can be used. For example, a DRPC can be identified by an internal representation of the call string. In this case, the DRPC can be executed by an optimized low level system interface using the internal representation. Thus, the call can be invoked without reconstructing an DRPC call in its entirety.

If the DRPC is identified and executed using a conventional DRPC call string, the reconstructed call string for the place_order_remote procedure call in the calls table in Figure 10 is: place_order_remote (Widget, DbA, 10, 50, filled). If the remote site is included in the call, the call is: place_order_remote @DbB (Widget, DbA, 10, 50, filled).

Referring to Figure 11A, at processing block 1122, a DRPC call string is reconstructed using the procedure identifier and parms fields from the calls table. The parameters for a call are reconstruct using the parms field in the calls table. Figure 13 illustrates a process flow for parsing the parms field. At

processing block 1302, a byte is extracted from the parms field. This byte represents the type of the current parameter in the parms field.

At decision block 1304 (i.e., "Type = 0?"), if the byte extracted from the parms field is equal to a termination value, processing ends at block 1306. If the byte is not a terminating value, processing continues at processing block 1308 to set "Len" to the next to bytes of the parms field. At processing block 1310, "Value" is set to the next "Len" bytes from the parms field. At processing block 1312, the contents of "Value" and "Len" are passed to a call constructor facility to incorporate this parameter information in the reconstructed call. Processing continues at block 1302 to process any remaining parameters in the parms field.

Referring to Figure 11A, at processing block 1124, the procedure is executed at the remote site. At decision block 1126 (i.e., "successful execution?"), if the procedure was successfully executed, processing continues at processing block 1127 to delete the selected call nodes table entry. Processing continues at decision block 1110 to check for additional calls to process.

If the procedure's execution was unsuccessful, the modifications made since the savepoint previously created are undone at processing block 1128. At processing block 1130, the Exceptions is invoked to create an entry in the Exceptions table to retain information concerning the unsuccessful completion of the procedure. The exception table can be stored at any location (e.g., originating, destination, or both sites). In the preferred embodiment, the exceptions table is stored at the destination site. Processing continues at decision block 1104 with any remaining transactions.

-- Exception Process Flow --

An exception can be stored in a relation that is located on either the originating or destination sites, or both. Figure 12 illustrates a process flow for storing an exception. At block 1202, entries are created in the error and transaction tables in the destination replication tables based on the values in the originating site's tables. At processing block 1204, entries are created in the calls and call nodes table in the destination replication tables based on the values in the originating site's tables. At block 1206, processing ends.

Figure 11A illustrated a method for executing transactional DRPC entries contained in the replication tables. Other methods can be used to process the replication table entries using replication capabilities of the present invention. For example, the present invention provides the ability to execute non-transactional DRPC entries contained in the replication tables. Figure 11B illustrates a method for executing non-transaction DRPC entries.

At processing block 1152, the non-transactional DRPCs to be executed are selected. At decision block 1154 (i.e., "all selected calls processed?"), if all of the DRPCs selected have been processed, processing ends at block 1174. If there are calls remaining, processing continues at processing block 1156 to select the next call and identify it by its call identifier. As in Figure 11A, the processing of a non-transactional DRPC in Figure 11B includes establishing a savepoint (at block 1158), constructing an DRPC call (at block 1160), and executing the DRPC at a remote site (at block 1162). If the execution is not successful, the changes since the savepoint are undone (at block 1172) and an exception is raised (at block 1174). If the execution is successful and no exceptions are raised during the execution, the record for this destination is deleted from the call nodes

table (at block 1168), and processing continues at decision block 1169. At decision block 1169 (i.e., "additional destinations for call?"), if there are additional destinations for the current call, processing continues at decision block 1154 to processing any remaining calls. If there are no additional destinations for call, the modifications are committed at block 1170.

Conflicts

The present invention provides the ability to identify conflicting modifications. For example, updates that have occurred to a remote copy of the data may be lost, if the replicated modification overwrites the current values in the remote copy. Thus, it is important to detect any conflicts. Further, if a conflict is detected, the present invention provides the ability to communicate an exception, to rollback any changes to a data copy after an exception is detected, and to incorporate exception handling in an application program. Exceptions and conflict information can be stored at the originating site, the destination site, or both.

– Conflict Detection –

As stated previously, a row-level replicated deferred remote procedure call has both the old and new values as part of its parameters. Thus, a potential conflict can be detected by comparing the old values of the row at the original updating site with the current value of the row at the targeted site for the replicated modification. If the values are different, a conflict exists between the local and remote data copies.

As previously illustrated the present invention provides the ability for an application, such as the order processing example described herein, to include error detection. As illustrated in the order processing example, the place_order_remote process includes examination of the local and remote copies of the order status. Thus, the present invention provides the ability for an application to identify conflicts within one of its procedures.

-- Treatment of Conflicts --

When a conflict is detected, information regarding the conflict can be identified and stored in the exceptions table (see Figure 3). The fields of the exception table provide an error code and a descriptive error string. In addition, the exceptions tables provides keys into other replications tables. This provides the ability to access the information stored in the replication tables associated with a DRPC in which the exception is raised.

For example, as a key into the transactions table, the exceptions table can contain a transaction identifier that corresponds to the current transaction being processed. To access the related entry in the calls table, the exceptions table further contains the call identifier of the current calls table entry being processed and the destination node (i.e., remote copy location). The calls table can be accessed using a procedure's call_id.

In addition to retaining information regarding a conflict, the present invention provides the ability to undo a deferred transaction such that all updates deferred by the original transaction are undone. The execution of deferred calls is contingent upon successful commit of a deferring transaction.

If the deferring transaction is rolled back, the deferred calls' queue encoded in the replication is rolled back.

The present invention further provides the ability to incorporate error handling in an application. Thus, an error can be handled as soon as an error is detected, or deferred for later processing. The exceptions table provides the ability to address any errors after an application's normal processing. The subsequent error processing can be done with various degrees of operator intervention and automation. The present invention provides the flexibility to allow an application to address the type of error handling.

Multiple conflict routines can be supplied to be used to resolve a conflict when it arises. They can be called in order until one of them returns a successful return value. If none of the resolution routines are successful, the exception is retained as an exception.

Thus, a method and apparatus for data replication has been provided.

CLAIMS

1. A method of replicating data modifications in one or more computer systems comprising the steps of:

storing, in a first computer system, a first copy of a data item;

storing, in a second computer system, a second copy of a data item;

performing a first modification to said first copy of said data item;

storing information about said modification such that said modification can be applied to said second copy of said data item; and

using said information about said first modification to perform a second modification to said second copy of said data item.

2. The method of claim 1 further including the step of identifying any exceptional occurrence while performing said first modification or said second modification.

3. The method of claim 2 further wherein said exceptional occurrence is a conflict identified when performing said second modification.

4. The method of claim 1 wherein said step of storing said information about said modification in said relations further includes the steps of:

retaining information regarding logical unit of work in a transactions relation;

retaining information regarding the destinations at which said logical unit of work is to be performed in a transaction nodes relation;

retaining information regarding procedures for replicating said logical unit of work;

retaining information regarding one or more destinations at which said procedures are to be performed; and

retaining information regarding exceptional conditions raised during said first and second modifications in an exceptions relation.

5. The method of claim 4 wherein said step of retaining information in said transactions relation further comprises the steps of:

providing a logical unit identifier in said transactions relation such that said logical unit identifier uniquely identifies each occurrence of said logical unit in said transactions relation;

providing an ordering value for each occurrence in said transactions relation such that said ordering value identifies the order in which modifications associated with each occurrence were applied to said data item;

providing a time value for each occurrence in said transactions relation such that said time value identifies the time at which modifications associated with each occurrence were applied to said data item; and

providing a user identifier for each occurrence in said transactions relation, said user identifier identifies the user invoking the modifications applied to said data item.

6. The method of claim 4 wherein said step of retaining information in said transaction nodes relation further comprises the steps of:

providing a logical unit identifier in said transaction nodes relation, said logical unit identifier identifies each occurrence of said logical unit in said transaction nodes relation; and

providing a destination identifier in said transaction nodes relation, said destination identifier uniquely identifies each location of said data item to be modified.

7. The method of claim 4 wherein a transaction identifier and destination identifier pair uniquely identifies each occurrence in said transaction nodes relation.

8. The method of claim 4 wherein said step of retaining information regarding procedures further comprises the steps of:

storing said information regarding procedures in a calls relation;

providing a calls relation entry identifier in said calls relation that uniquely identifies an entry in said calls relation;

providing a logical unit identifier in said calls relation that uniquely identifies each occurrence of said logical unit in said calls relation;

providing a procedure identifier associated with each occurrence in said calls relation that identifies a logical series of steps for modifying said data item; and

providing a parameter count value for each occurrence in said calls relation such that said parameter count value identifies the informational items used to complete said logical series of steps.

9. The method of claim 8 further including the step of:

providing parameter information that identifies a plurality of arguments for said each of occurrence.

10. The method of claim 9 wherein said parameter information is stored in said calls relation.

11. The method of claim 9 wherein said parameter information is stored in a parameters table.

12. The method of claim 11 wherein said parameters table further includes said call relation entry identifier and said logical unit identifier.

13. The method of claim 4 wherein said step of retaining information in said calls relation further comprises the steps of:

providing a calls relation entry identifier in a parameters relation, said calls relation entry identifier associates each entry in said parameters relation with a logical series of steps; and

providing a parameter ordering value, said parameter ordering value identifies the order in which each entry associated with one of said logical series of steps are listed in a string identifying said one of said logical series of steps.

14. The method of claim 4 wherein said step of retaining information in said exceptions relation further comprises the steps of:

providing a logical unit identifier in said exceptions relation, said logical unit identifier identifies each occurrence of said logical unit in said exceptions relation;

providing a destination identifier in said exceptions relation, said destination identifier uniquely identifies each location of said data item to be modified;

providing a code value to identify an exceptional occurrence while performing said first modification or said second modification; and

providing a descriptive value to identify said exceptional occurrence while performing said first modification or said second modification.

15. The method of claim 4 wherein said step of retaining information regarding said one or more procedure destinations further including the steps of:

storing a mapping identifier in said relations to identify a mechanism for mapping said procedures to said one or more procedure destinations.

16. The method of claim 15 wherein said mapping mechanism is a call nodes relation, said call nodes table uniquely identifies an occurrence of a procedure in said calls relation and a destination at which said occurrence of a procedure is to be performed.

17. A method of replicating a value-oriented change to copies of data in one or more computer systems comprising the steps of:

providing a first computer system containing a first copy of said data;

providing a second computer system containing a second copy of said data; and

associating a procedure with said first copy of said data such that said procedure is invoked when a modification is made to said first copy of said data, and said procedure causes information associated with said modification to be stored in relations.

18. The method of claim 17 further including the steps of:

accessing said information associated with said modification; and

applying said modification to said second copy of said data.

19. The method of claim 17 wherein said modification is an update and said application of said modification to said second copy of said data consists of replacing the current value of said second copy of said data with a new value of said first copy of said data.

20. The method of claim 17 wherein said modification is an update and said application of said modification to said second copy of said data consists of:

comparing an old value of said first data with a new value of said first copy said data to identify changed data in said first copy of said data;

replacing a current value of said changed data in said second copy of said data with a new value of said first copy of said data.

21. The method of claim 17 wherein said modification is a delete and said application of said modification to said second copy of said data consists of deleting said second copy of said data.

22. The method of claim 17 wherein said modification is an insert and said application of said modification to said second copy of said data consists of creating said second copy of said data, said second copy of said data having the same value as said first copy of said data.

23 The method of claim 17 further including the steps of:

identifying any exceptional occurrence while performing said modification; and

retaining information regarding said any exceptional occurrence in said relations.

24. The method of claim 23 wherein the step of identifying said any exceptional occurrence further comprises the steps of:

comparing an old value of said first copy of said data with a new value of said first copy of said data to identify changed data;

comparing an old value of said changed data with a current value of said changed data in said second copy of said data; and

identifying an exception when said old value is not equal to said current value.

25. The method of claim 23 wherein the step of identifying said any exceptional occurrence further comprises the steps of:

comparing an old value of said first data with a current value of said second data; and

identifying an exception when said old value is not equal to said current value.

26. A method of replicating a logic-oriented modification to copies of data in one or more computer systems comprising the steps of:

providing a first computer system containing a first copy of said data;

providing a second computer system containing a second copy of said data;

modifying said first copy of said data with a procedure containing a series of steps for modifying said data; and

retaining information about said procedure in relations.

27. The method of claim 26 further including the steps of:

accessing said information about said procedure; and

executing said procedure to modify said second copy of said data.

28. The method of claim 26 further including the step of identifying any exceptional occurrence while performing said procedure.

29. The method of claim 28 wherein said step of identifying said any exceptional occurrence is incorporated in said procedure such that said procedure defines the circumstances that raise said exceptional occurrences.

30. The method of claim 28 further including the step of addressing said any exceptional occurrence.

31. The method of claim 30 wherein said step of addressing said any exceptional occurrence is incorporated in said procedure such that said procedure handles said any exceptional occurrence.

32. The method of claim 30 wherein said step of addressing said any exceptional occurrence is done by said procedure during execution of said procedure, and said information associated with said any exceptional occurrence is re...ned in said relations to be addressed after said execution of said procedure.

33. A method of replicating a value-oriented change to copies of data in one or more computer systems comprising the steps of:

providing a first computer system containing a first copy of said data;

providing a second computer system containing a second copy of said data; and

associating a plurality of procedures with said first copy of said data and said second copy of said data;

invoking one of said plurality of procedures when a modification is made to an associated copy of said data;

determining whether said modification is of said first copy of said data; and

storing information associated with said modification when said modification is of said first copy of said data.

34. The method of claim 33 wherein said step of determining whether said modification is of said first copy of said data further comprises the steps of:

setting a replication flag set to true when said modification is made to said first copy of said data;

storing information associated with said modification when said replication flag is false; and

disregarding said information associated with said replication flag is true.

35. The method of claim 33 wherein said step of determining whether said modification is of said first copy of said data further comprises the steps of:

associating a first user identifier with said modification when said modification is of said first copy of said data;

associating a second user identifier with said modification when said modification is of said second copy of said data;

storing information associated with said modification when a user identifier associated with said modification is said first user identifier; and

disregarding said information associated with said modification when said user identifier associated with said modification is said second user identifier.

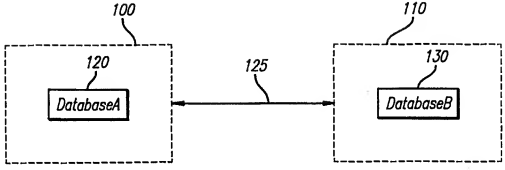

100
110
120
125
130
DatabaseA
DatabaseB

FIG. 1

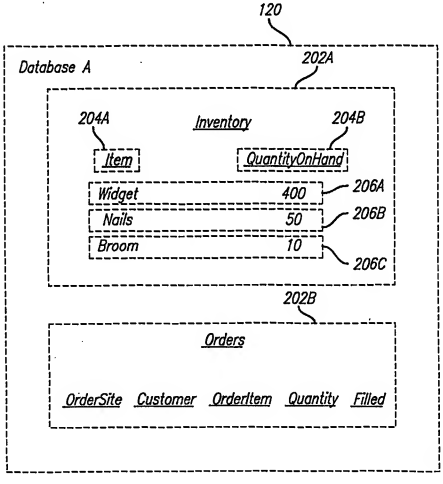

120
Database A
202A
204A
Inventory
204B
Item
QuantityOnHand
Widget 400
206A
Nails 50
206B
Broom 10
206C
202B
Orders
OrderSite Customer OrderItem Quantity Filled

FIG. 2A

120
Database A
Inventory
Item
QuantityOnHand
Widget
400
Nails
50
Broom
10
Orders
OrderSite
Customer
OrderItem
Quantity
Filled
130
Database B
Inventory
Item
QuantityOnHand
Widget
400
Nails
50
Broom
10
Orders
OrderSite
Customer
OrderItem
Quantity
Filled
Basic_Order_Transaction:
1. Place Order:
a. Insert entry in Orders Table where:
OrderSite = database site at which order is entered (i.e., DB_A or DB_B)
Customer = customer placing order
Order_Item = inventory item ordered
Quantity = quantity of inventory item ordered
OrderSite = database site at which order is entered (i.e., DB_A or DB_B)
Filled = "Y", if qoh > quantity ordered, or "B", otherwise.
2. Inventory_Check:
a. Access QuantityOnHand (qoh) from Inventory Table;
b. If qoh > quantity ordered, update qoh in Inventory Table such that qoh = qoh - quantity ordered
3. Commit (i.e., make changes permanent)

FIG. 2B

FIG. 2C

Basic_Order_Transaction:

1. Place Order:
   a. Insert entry in Orders Tables where:
      OrderSite = database site at which order is entered (i.e., DB_A or DB_B)
      Customer = customer placing order
      Order_Item = inventory item ordered
      Quantity = quantity of inventory item ordered
      OrderSite = database site at which order is entered (i.e., DB_A or DB_B)
      Filled = "F", if qoh > quantity ordered, or "B", otherwise.
2. Inventory_Check:
   a. Access QuantityOnHand (qoh) from Inventory Table;
   b. If qoh > quantity ordered, update qoh in Inventory Table such that qoh = qoh – quantity ordered
3. Commit (i.e., make changes permanent)

Before Replication
Database A
Inventory
Item
QuantityOnHand
Widget
350
Nails
50
Broom
10
Orders
OrderSite
Customer
OrderItem
Quantity
Filled
DB_A
10
Widget
50
F
Database B
Inventory
Item
QuantityOnHand
Widget
310
Nails
50
Broom
10
Orders
OrderSite
Customer
OrderItem
Quantity
Filled
DB_B
11
Widget
40
F
After DbB => DbA Replication
Database A
Inventory
Item
QuantityOnHand
Widget
310
Nails
50
Broom
10
Orders
OrderSite
Customer
OrderItem
Quantity
Filled
DB_A
10
Widget
50
F
DB_B
11
Widget
40
F
Database B
Inventory
Item
QuantityOnHand
Widget
360
Nails
50
Broom
10
Orders
OrderSite
Customer
OrderItem
Quantity
Filled
DB_B
11
Widget
40
F

FIG. 2D(1)

After DbA => DbB Replication

Database A

Inventory

Item	QuantityOnHand
Widget	310
Nails	50
Broom	10

Orders

OrderSite	Customer	OrderItem	Quantity	Filled
DB_A	10	Widget	50	F
DB_B	11	Widget	40	F

Database B

Inventory

Item	QuantityOnHand
Widget	310
Nails	50
Broom	10

Orders

OrderSite	Customer	OrderItem	Quantity	Filled
DB_B	11	Widget	40	F
DB_A	10	Widget	50	F

FIG. 2D(2)

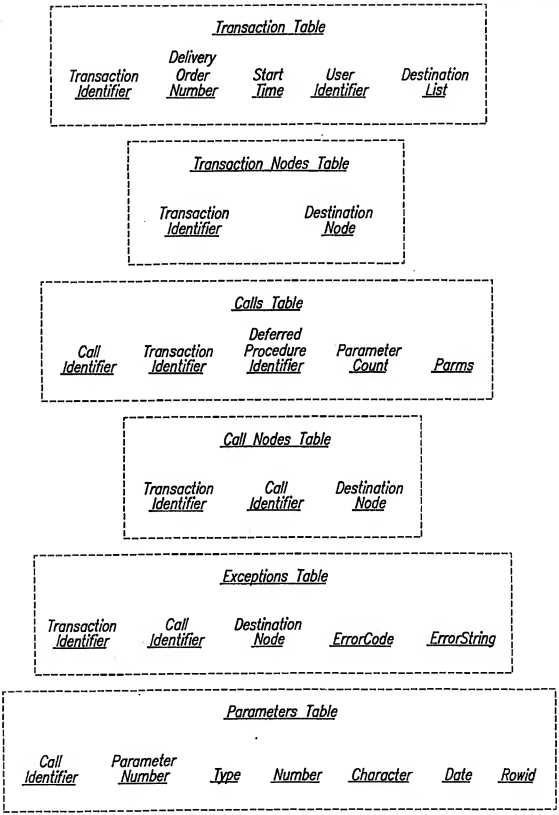

Transaction Table
Transaction Identifier
Delivery Order Number
Start Time
User Identifier
Destination List
Transaction Nodes Table
Transaction Identifier
Destination Node
Calls Table
Call Identifier
Transaction Identifier
Deferred Procedure Identifier
Parameter Count
Parms
Call Nodes Table
Transaction Identifier
Call Identifier
Destination Node
Exceptions Table
Transaction Identifier
Call Identifier
Destination Node
ErrorCode
ErrorString
Parameters Table
Call Identifier
Parameter Number
Type
Number
Character
Date
Rowid

FIG. 3

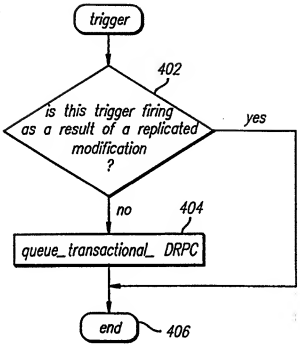

trigger
402
is this trigger firing as a result of a replicated modification ?
yes
no
404
queue_transactional_ DRPC
end
406

FIG. 4

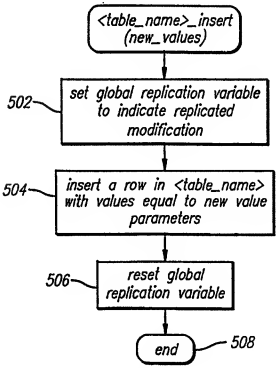

<table_name>_insert (new_values)
502
set global replication variable to indicate replicated modification
504
insert a row in <table_name> with values equal to new value parameters
506
reset global replication variable
end
508

FIG. 5A

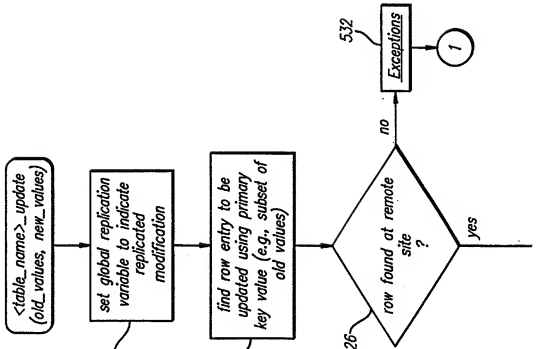

<table_name>_update (old_values, new_values)
set global replication variable to indicate replicated modification
522
find row entry to be updated using primary key value (e.g., subset of old values)
524
row found at remote site ?
526
no
yes
532
Exceptions
1

FIG. 5B(1)

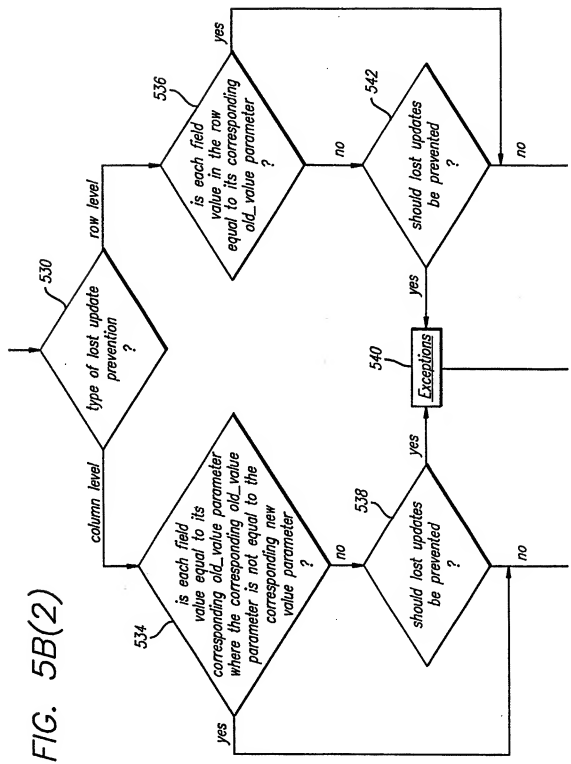

530
type of lost update prevention ?
column level
row level
534
is each field value equal to its corresponding old_value parameter where the corresponding old_value parameter is not equal to the corresponding new value parameter ?
yes
no
538
should lost updates be prevented ?
yes
no
540
Exceptions
536
is each field value in the row equal to its corresponding old_value parameter ?
yes
no
542
should lost updates be prevented ?
yes
no

FIG. 5B(2)

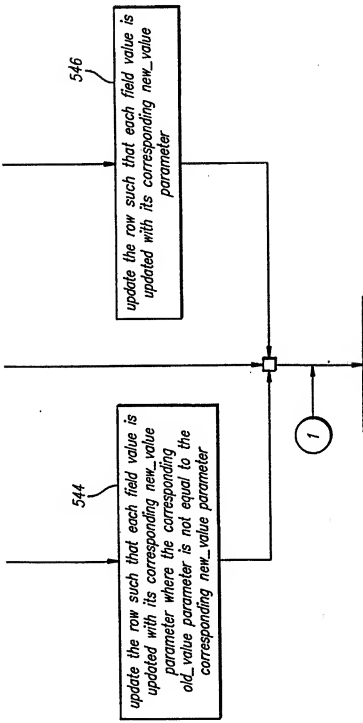

546
update the row such that each field value is updated with its corresponding new_value parameter
544
update the row such that each field value is updated with its corresponding new_value parameter where the corresponding old_value parameter is not equal to the corresponding new_value parameter
1
548
reset global replication variable
550
end

FIG. 5B(3)

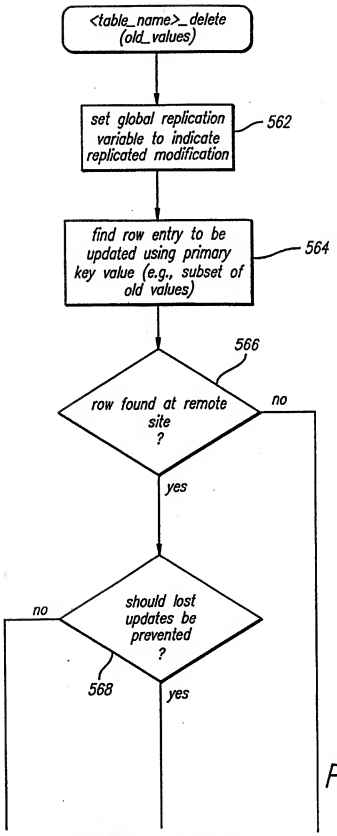

<table_name>_delete (old_values)
set global replication variable to indicate replicated modification
562
find row entry to be updated using primary key value (e.g., subset of old values)
564
566
row found at remote site ?
no
yes
should lost updates be prevented ?
no
568
yes

FIG. 5C(1)

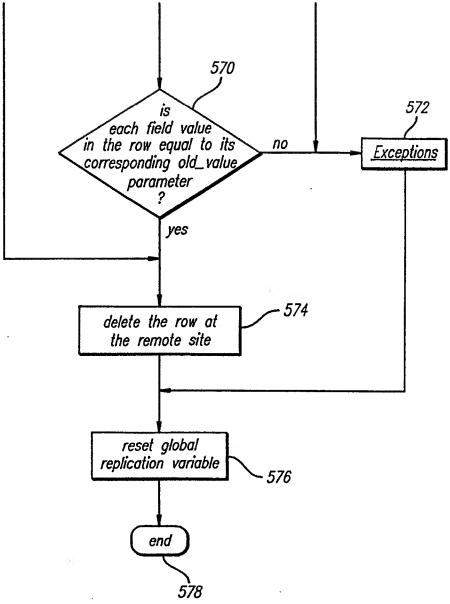
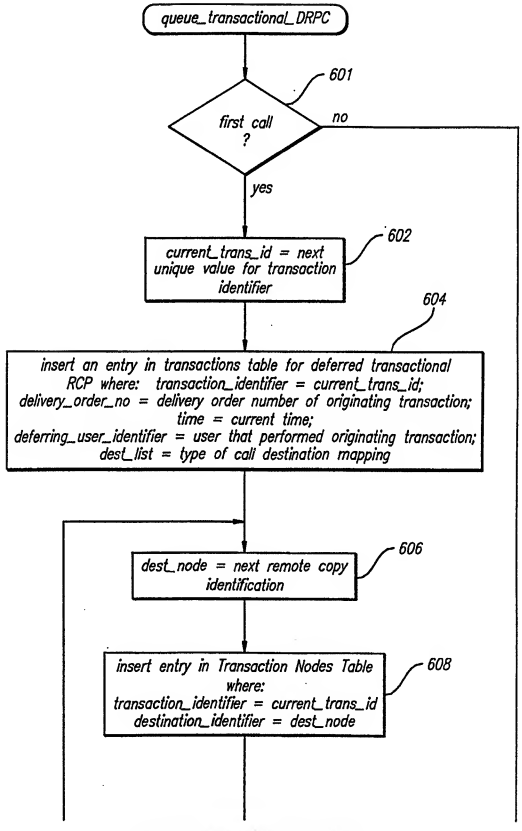

570
is each field value in the row equal to its corresponding old_value parameter ?
no
572
Exceptions
yes
delete the row at the remote site
574
reset global replication variable
576
end
578

FIG. 5C(2)

FIG. 6(a)


queue_transactional_DRPC
601
first call ?
no
yes
current_trans_id = next unique value for transaction identifier
602
604
insert an entry in transactions table for deferred transactional RCP where: transaction_identifier = current_trans_id; delivery_order_no = delivery order number of originating transaction; time = current time; deferring_user_identifier = user that performed originating transaction; dest_list = type of call destination mapping
606
dest_node = next remote copy identification
608
insert entry in Transaction Nodes Table where: transaction_identifier = current_trans_id destination_identifier = dest_node

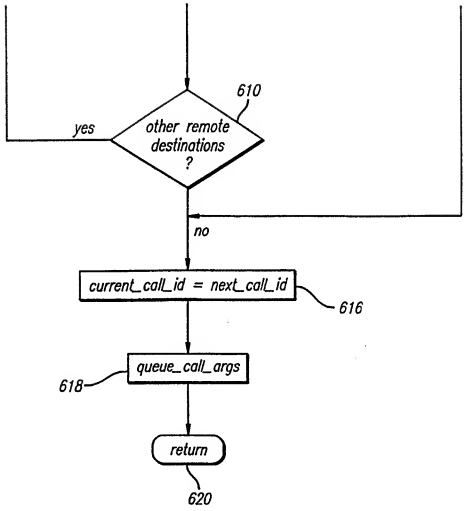

610
other remote destinations ?
yes
no
current_call_id = next_call_id
616
queue_call_args
618
return
620

FIG. 6(b)

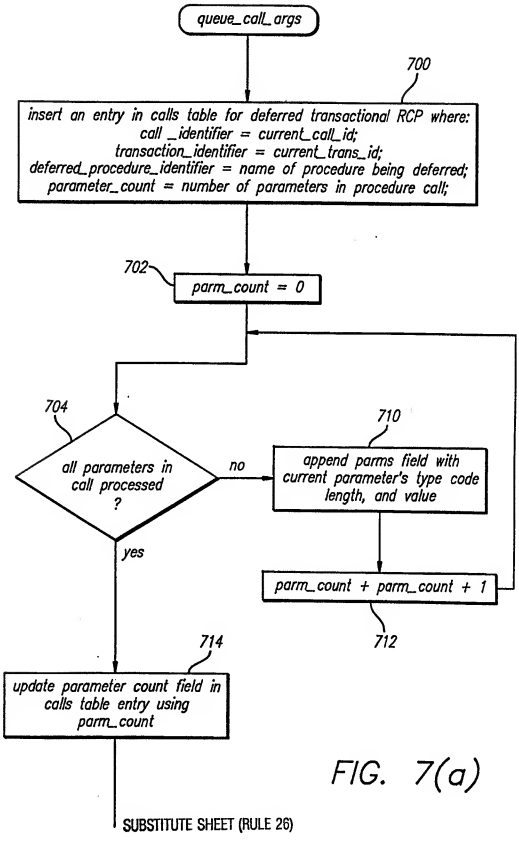

queue_call_args
700
insert an entry in calls table for deferred transactional RCP where:
call _identifier = current_call_id;
transaction_identifier = current_trans_id;
deferred_procedure_identifier = name of procedure being deferred;
parameter_count = number of parameters in procedure call;
702
parm_count = 0
704
all parameters in call processed ?
no
710
append parms field with current parameter's type code length, and value
yes
parm_count + parm_count + 1
712
714
update parameter count field in calls table entry using parm_count

FIG. 7(a)

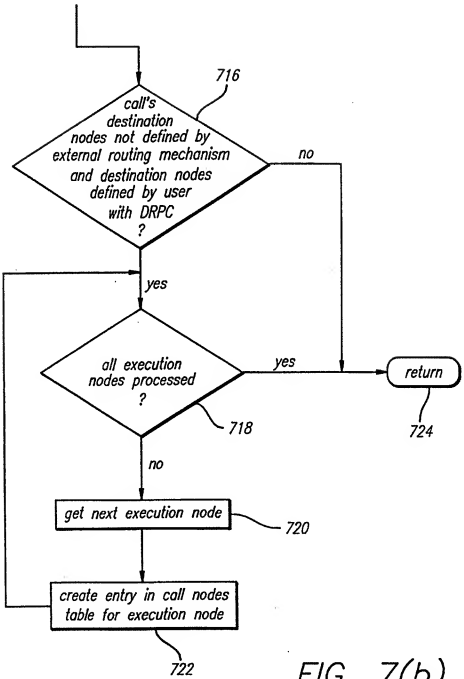
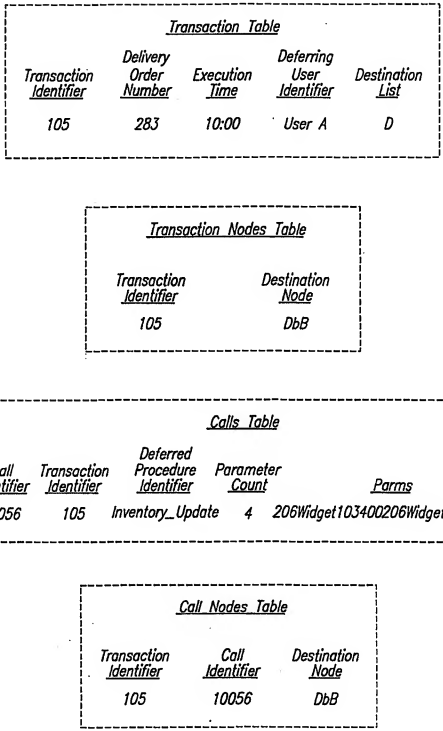

716
call's destination nodes not defined by external routing mechanism and destination nodes defined by user with DRPC ?
no
yes
all execution nodes processed ?
yes
return
718
724
no
get next execution node
720
create entry in call nodes table for execution node
722

FIG. 7(b)



Transaction Table

Transaction Identifier	Delivery Order Number	Execution Time	Deferring User Identifier	Destination List
105	283	10:00	User A	D

Transaction Nodes Table

Transaction Identifier	Destination Node
105	DbB

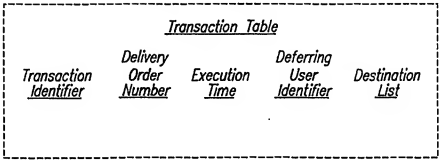
Calls Table

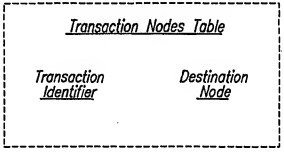
Call Identifier	Transaction Identifier	Deferred Procedure Identifier	Parameter Count	Parms
10056	105	Inventory_Update	4	206Widget103400206Widget1033500

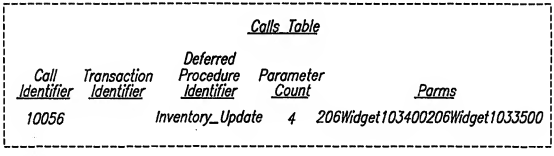
Call Nodes Table

Transaction Identifier	Call Identifier	Destination Node
105	10056	DbB

FIG. 8A


Transaction Table
Transaction Identifier
Delivery Order Number
Execution Time
Deferring User Identifier
Destination List


Transaction Nodes Table
Transaction Identifier
Destination Node


Calls Table
Call Identifier
Transaction Identifier
Deferred Procedure Identifier
Parameter Count
Parms
10056
Inventory_Update
4
206Widget103400206Widget1033500

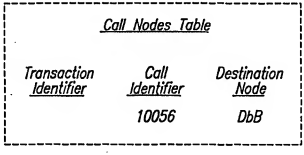
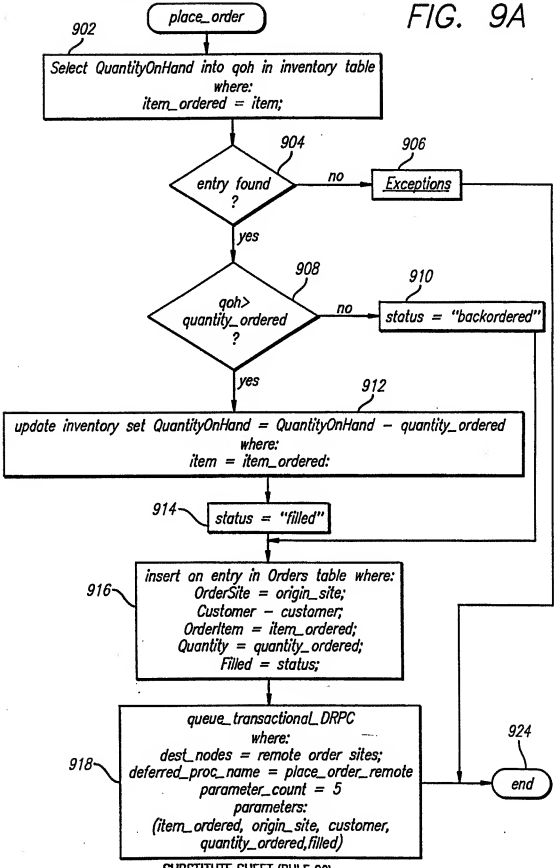

Call Nodes Table
Transaction Identifier
Call Identifier
Destination Node
10056
DbB

FIG. 8B

FIG. 9A


place_order
902
Select QuantityOnHand into qoh in inventory table
where:
item_ordered = item;
904
entry found
?
no
906
Exceptions
yes
908
qoh>
quantity_ordered
?
no
910
status = "backordered"
yes
912
update inventory set QuantityOnHand = QuantityOnHand − quantity_ordered
where:
item = item_ordered:
914
status = "filled"
916
insert an entry in Orders table where:
OrderSite = origin_site;
Customer − customer;
OrderItem = item_ordered;
Quantity = quantity_ordered;
Filled = status;
queue_transactional_DRPC
where:
dest_nodes = remote order sites;
deferred_proc_name = place_order_remote
parameter_count = 5
parameters:
(item_ordered, origin_site, customer,
quantity_ordered,filled)
918
924
end

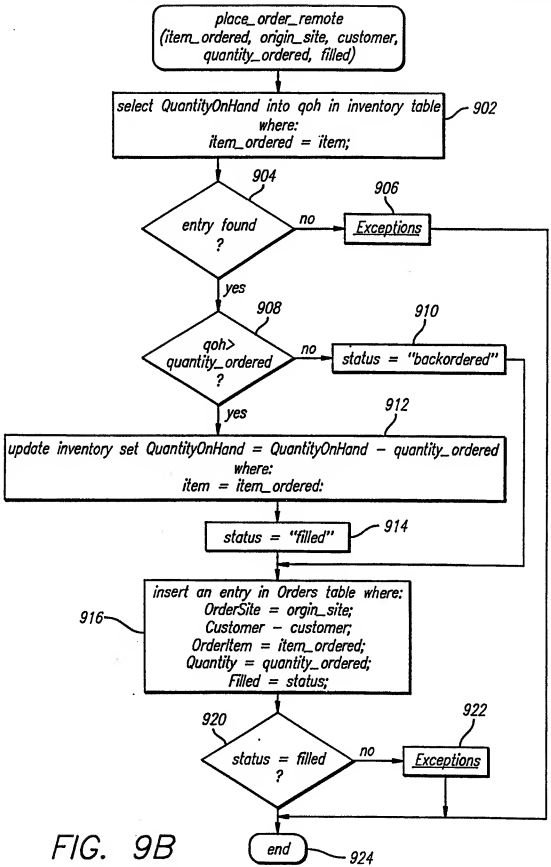

place_order_remote
(item_ordered, origin_site, customer, quantity_ordered, filled)
select QuantityOnHand into qoh in inventory table where:
item_ordered = item;
902
904
entry found ?
no
906
Exceptions
yes
908
qoh> quantity_ordered ?
no
910
status = "backordered"
yes
912
update inventory set QuantityOnHand = QuantityOnHand – quantity_ordered where:
item = item_ordered:
status = "filled"
914
insert an entry in Orders table where:
OrderSite = orgin_site;
Customer – customer;
OrderItem = item_ordered;
Quantity = quantity_ordered;
Filled = status;
916
920
status = filled ?
no
922
Exceptions
end
924

FIG. 9B

Transaction Table

Transaction Identifier	Delivery Order Number	Commit Time	Deferring User Identifier	Destination List
4	10	1:00	User A	D

Transaction Nodes Table

Transaction Identifier	Destination Node
4	DbB

Calls Table

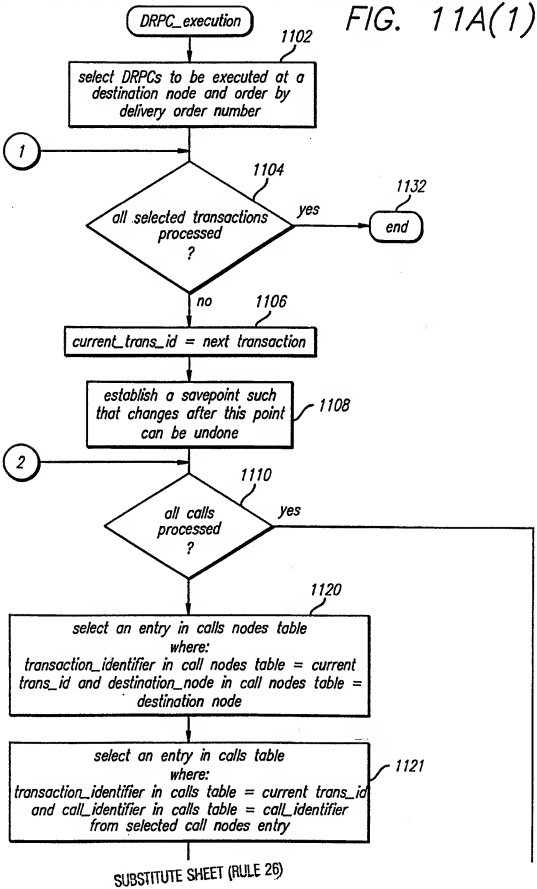
Call Identifier	Transaction Identifier	Deferred Procedure Identifier	Parameter Count	Parms
1	4	place_order_remote	4	206Widget203DbA10250102502O6filled0

Call Nodes Table

Transaction Identifier	Call Identifier	Destination Node
4	1	DbB

FIG. 10

FIG. 11A(1)


DRPC_execution
1102
select DRPCs to be executed at a destination node and order by delivery order number
1
1104
all selected transactions processed ?
yes
1132
end
no
1106
current_trans_id = next transaction
establish a savepoint such that changes after this point can be undone
1108
2
1110
all calls processed ?
yes
1120
select an entry in calls nodes table where: transaction_identifier in call nodes table = current trans_id and destination_node in call nodes table = destination node
select an entry in calls table where: transaction_identifier in calls table = current trans_id and call_identifier in calls table = call_identifier from selected call nodes entry
1121

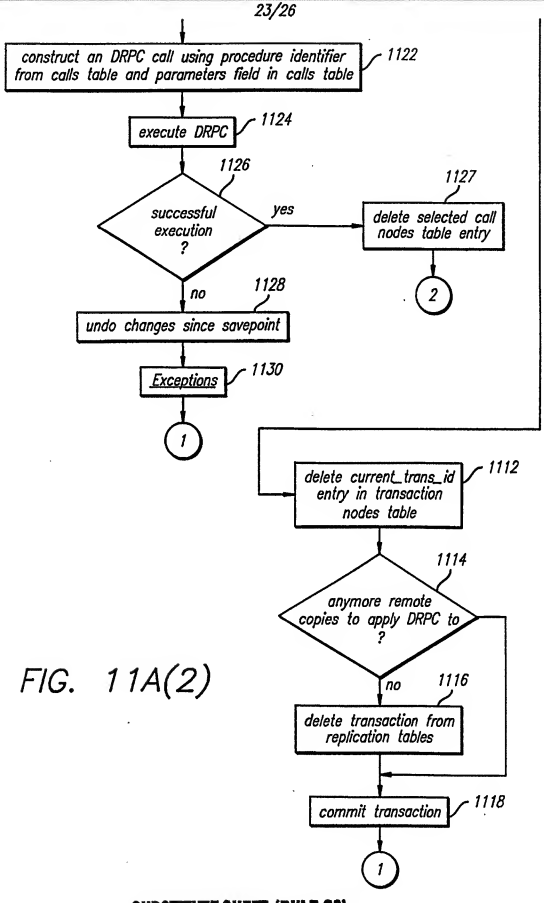

construct an DRPC call using procedure identifier from calls table and parameters field in calls table
1122
execute DRPC
1124
1126
successful execution ?
yes
1127
delete selected call nodes table entry
2
no
1128
undo changes since savepoint
Exceptions
1130
1
delete current_trans_id entry in transaction nodes table
1112
1114
anymore remote copies to apply DRPC to ?
no
1116
delete transaction from replication tables
commit transaction
1118
1

FIG. 11A(2)

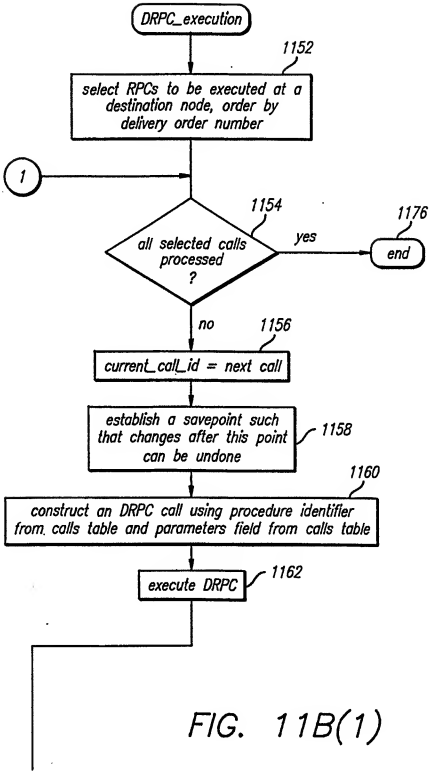

DRPC_execution
1152
select RPCs to be executed at a destination node, order by delivery order number
1
1154
all selected calls processed ?
yes
1176
end
no
1156
current_call_id = next call
establish a savepoint such that changes after this point can be undone
1158
1160
construct an DRPC call using procedure identifier from calls table and parameters field from calls table
execute DRPC
1162

FIG. 11B(1)

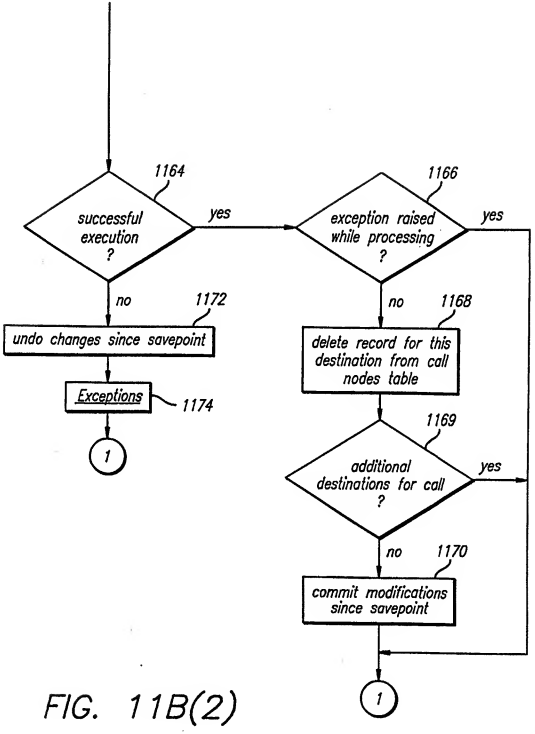

1164
successful execution ?
yes
1166
exception raised while processing ?
yes
no
1172
undo changes since savepoint
Exceptions
1174
1
no
1168
delete record for this destination from call nodes table
1169
additional destinations for call ?
yes
no
1170
commit modifications since savepoint
1

FIG. 11B(2)

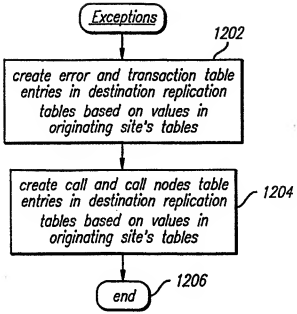

Exceptions
1202
create error and transaction table entries in destination replication tables based on values in originating site's tables
create call and call nodes table entries in destination replication tables based on values in originating site's tables
1204
end
1206

FIG. 12

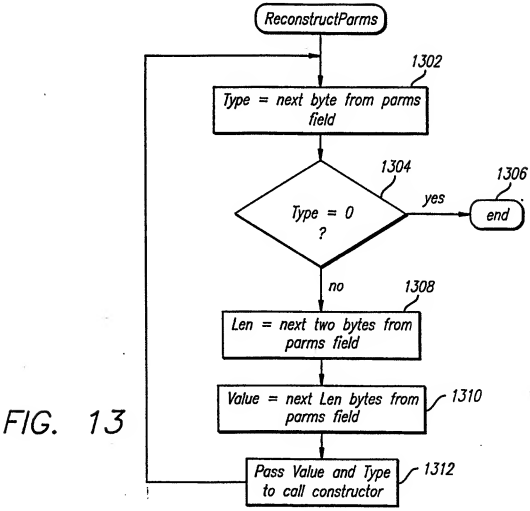

ReconstructParms
1302
Type = next byte from parms field
1304
Type = 0 ?
yes
1306
end
no
1308
Len = next two bytes from parms field
Value = next Len bytes from parms field
1310
Pass Value and Type to call constructor
1312

FIG. 13